A Genetic Algorithm Framework Grounded in Biology

Joshua ZR Lim[1], Zhen Qin Aw[1], Desmond JW Goh[1], Jian Ann How[1], Samuel XZ Low[1], Bryan ZL Loo[1], Maurice HT Ling[1,2,3]
[1]School of Chemical and Life Sciences, Singapore Polytechnic, Singapore
[2]Department of Zoology, The University of Melbourne, Australia
[3]Corresponding email: mauriceling@acm.org

Abstract
Genetic algorithm (GA) is a heuristic search method inspired by biological evolution of genetic organisms by optimizing the genotypic combinations encoded within each individual with the help of evolutionary operators, such as reproduction, mutation and cross-over. This manuscript aims to present a simple GA framework written in Python programming language that conforms to biological hierarchy starting from gene to chromosome to genome (as organism) to population. Hence, we believe that this framework may be useful in both education and biological simulation on top of the usual domains where GA were used.

1. Description

Genetic algorithm (GA) is a heuristic search method inspired by biological evolution of genetic organisms (Engelbrecht, 2007) which was first suggested by Fraser (Fraser, 1957a; Fraser, 1957b) and later, popularized by Holland (1975). Since then, GA had been used successfully in a number of applications (see Shiekh et al. (2008) for a review).

In order to model biological evolution, the characteristics of individual organisms are encoded in a string-based chromosome as genotypes. The goal of GA is to optimize the combination of genotypes as calculated by a fitness function. Evolutionary operators, such as mutation and cross-over, are the primary means to optimize the genotypic combinations from one generation to the next. This is supported by reproduction selection operators which decide which individuals are chosen to propagate the next generation. Whitley (1994) and Srinivas and Patnaik (2004) presented excellent tutorials on GA.

This manuscript aims to present a simple GA framework written in Python programming language that conforms to biological hierarchy starting from gene to chromosome to genome (as organism) to population, as well as incorporating biological and ecological relevance. For example, it is possible to simulate mutational hotspots at the genetic level, simulating vitality and age at the organism level, and infrequent environmental catastrophe that can eliminate large numbers of organisms at the population level. Other notable Python GA frameworks are pyevolve[1], genalgy[2], and pygene[3].

[1] http://pyevolve.sourceforge.net/
[2] http://hobbiton.thisside.net/genetic/
[3] http://www.freenet.org.nz/python/pygene/

This framework consists of 3 classes; Chromosome, Organism, and Population; licensed under Python Software Foundation License version 2.

The data to be optimized is encoded as an alphanumeric string or any data structures (known as a base) as a chromosome. Our implementation allows for both chromosome-wide and region-specfic mutation (*Chromosome.rmutate* method) as well as mutation to only base of the chromosome (*Chromosome.kmutate* method). The former can be used to simulate background mutation and the biological equivalent of mutational hotspots while the latter can be used to simulate known mutations. In both cases, 6 types of mutations are defined, namely point mutation (in-place change of a base), insertion mutation (adding a base), deletion mutation (deleting a base), inversion (inverting a stretch of the chromosome), duplication (duplicating a stretch of the chromosome), translocation (moving a stretch of chromosome to another random position).

Although most GA implementations used one chromosome to represent the entire genotype of an organism Holland (1975), our implementation allows for segmentation of genotype to one or more chromosomes. At the same time, we considered the concept of biological age and developmental status of the organism. This will enable the simulation of age-regulated mutation rates. Therefore, we encapsulate the genome (comprising of one or more chromosome) and a status table (representing the physiological status of an individual) as an organism. Each organism is also responsible for the evaluation of its own fitness (*Organism.fitness* method) and performing mutations (*Organism.mutation_scheme* method) where additional mutation rate on top of background mutation rate can be defined. The rationale to implement these 2 methods at the organism level is to cater for applications that may require individually tailored fitness evaluation and mutation operations within a population of organisms. As these 2 methods are application-specific, they have to be over-ridden by the user.

A population can be defined as a set of organisms propagating/reproducing further generations (Melanie, 1999). This class manages mate selection, population control and reporting the status of each generation. This is done in a 6-step process. Firstly, with the exception of the initial generation, there will be a pre-mating population control (*Population.prepopulation_control* method) to simulate events before sexual maturation, such as childhood fatalities, or events leading to the unsuitability of the individual to propagate the next generation. Secondly, the matured population undergoes mating (*Population.mating* method). This may include mate selection based on fitness and status of each individual in the population and the actual process of mating. Thirdly, there will be a post-mating population control (*Population.postpopulation_control* method) to simulate events after mating. Fourthly, the remaining population will undergo their individual mutation based on the organism's mutation scheme. Fifthly, the population will undergo generational-specific events (*Population.generation_events* method) which may include simulation of environmental catastrophe or changes in mutation. Lastly, the status of the generation is reported (*Population.report* method). Hence, there are 3 locations where population control or any events within each generation can be simulated - pre-mating population control, post-mating population control and generational-specific events. This is to enable flexibility and segregation of events.

Our implementation also includes a method to fossilize part of or the entire population (*Population.freeze* method) for in-depth analysis, as well as a revival method (*Population.revive* method) to revive the entire fossilized population. The experimental equivalence of this in biology will be cryopreservation of cells and tissues. In addition, multiple populations with migration policies (Skolicki, 2005) may be used to construct island GA (ElNainay et al., 2009; King, 2004).

Section 3 of this manuscript demonstrates an example of initiating a population using *population_constructor* function which takes *population_data* dictionary as parameter and followed by *population_simulate* function to execute the GA. These 2 functions also act as templates where more complex organism and population makeup can be established.

2. Source Code

```
"""
Framework for Genetic Algorithm Applications.
Date created: 23rd February 2010
Licence: Python Software Foundation License version 2
"""
import random, os
from copy import deepcopy

class Chromosome(object):
    """Representation of a linear chromosome."""
    def __init__(self, sequence, base,
                 background_mutation=0.0001):
        """
        Sets up a chromosome.

        @param sequence: a subscriptable object (list or string) representing
            the sequence of the chromosome.
        @param base: a subscriptable object (list or string) representing
            allowable entities in the sequence.
        @param background_mutation: background mutation rate represented as the
            probability of number of mutations per base. Default = 0.0001
            (0.01%).
        """
        self.sequence = sequence
        self.base = base
        self.background_mutation = background_mutation

    def rmutate(self, type='point', rate=0.01, start=0, end=-1):
        """
        Random Mutation operator - to simulate random point, insertion,
        deletion, inversion, gene translocation and gene duplication events.

        The start and end parameters are useful for simulating mutational
        hotspots in the genome.

        @param type: type of mutation. Accepts 'point' (point mutation),
            'insert' (insert a base), 'delete' (delete a base), 'invert'
            (invert a stretch of the chromosome), 'duplicate' (duplicate a
            stretch of the chromosome), 'translocate' (translocate a stretch of
            chromosome to another random position). Default = point.
        @param rate: probability of mutation per base above background
            mutation rate. Default = 0.01 (1%). No mutation event will ever
            happen if (rate + background_mutation) is less than zero.
        @param start: starting base on the sequence for mutation.
            Default = 0, start of the genome.
```

```
    @param end: last base on the sequence for mutation. Default = -1, end of
        the genome.
    """
    if end > len(self.sequence) - 1: end = len(self.sequence) - 1
    if end == -1: end = len(self.sequence) - 1
    if start == end: start = 0
    length = int(end - start)
    mutation = int((self.background_mutation + rate) * length)
    while mutation > 0:
        position = int(start) + random.randrange(length - 1)
        new_base = self.base[random.randrange(len(self.base))]
        if type == 'point':
            self.sequence[position] = new_base
        if type == 'delete':
            self.sequence.pop(position)
        if type == 'insert':
            self.sequence.insert(position, new_base)
        if type == 'duplicate':
            end_pos = random.randrange(position + 1, end)
            fragment = self.sequence[position:end_pos]
            for i in range(len(fragment)):
                self.sequence.insert(end_pos + i, fragment[i])
        if type == 'invert':
            end_pos = random.randrange(position + 1, end)
            fragment = [self.sequence.pop(position)
                        for i in range(end_pos - position)]
            fragment.reverse()
            for base in fragment:
                self.sequence.insert(position, base)
        if type == 'translocate':
            end_pos = random.randrange(position + 1, end)
            fragment = [self.sequence.pop(position)
                        for i in range(end_pos - position)]
            insertion_point = random.randint(0, len(self.sequence))
            for i in range(len(fragment)):
                self.sequence.insert(insertion_point + i, fragment[i])
        mutation = mutation - 1

def kmutate(self, type='point', start=0, end=0, sequence=None, tpos=0):
    """
    Known Mutation operator - to simulate a known point, insertion,
    deletion, inversion, gene translocation or gene duplication event.

    The required parameters will be determined by the type of mutation:
        - point requires
            - start (the position of the base to mutate)
            - sequence (the new base)
        - delete requires
            - start (the position of the base to delete)
        - insert requires
            - start (the position of the base to begin insertion)
            - sequence (list of sequence or a base to insert)
        - invert requires
            - start (the position of the base to start inversion)
            - end (the position of the base to end inversion)
        - duplicate requires
            - start (the position of the starting base to duplicate)
            - end (the position of the last base to duplicate)
        - translocate requires
            - start (the position of the base to start translocation)
            - end (the position of the base to end translocation)
            - tpos (the position of the base insert the translocated
                sequence)
```

```
        @param type: type of mutation. Accepts 'point' (point mutation),
            'insert' (insert a base), 'delete' (delete a base), 'invert'
            (invert a stretch of the chromosome), 'duplicate' (duplicate a
            stretch of the chromosome), 'translocate' (translocate a stretch of
            chromosome to another random position). Default = point.
        @param start: starting base on the chromosome for mutation.
            Default = 0, start of the genome.
        @param end: last base on the chromosome for mutation. Default = 0.
        @param sequence: list of bases to change to (point mutation) or to
            insert (insertion mutation)
        @param tpos: position of the chromosome to insert the translocated
            sequence.
        """
        if type == 'point':
            self.sequence[start] = sequence
        if type == 'delete':
            self.sequence.pop(start)
        if type == 'insert':
            for base in list(sequence):
                self.sequence.insert(start, base)
        if type == 'duplicate':
            fragment = self.sequence[start:end + 1]
            for i in range(len(fragment)):
                self.sequence.insert(end + i, fragment[i])
        if type == 'invert':
            for base in self.sequence[start:end]:
                self.sequence.insert(start, base)
        if type == 'translocate':
            fragment = [self.sequence.pop(start)
                        for i in range(end - start)]
            for i in range(len(fragment)):
                self.sequence.insert(tpos + i, fragment[i])

    def replicate(self):
        """
        Replicates (deep copy) the chromosome.

        @return: a copy of current chromosome.
        """
        return deepcopy(self)


class Organism(object):
    """
    An organism represented by a list of chromosomes and a status table. This
    class should almost never be instantiated on its own but acts as an ancestor
    class as some functions need to be over-ridden in the inherited class for
    the desired use.

    Methods to be over-ridden in the inherited class or substituted are
        - fitness
        - mutation_scheme

    Pre-defined status are
        1. alive - is the organism alive? True or False.
        2. vitality - percentage of maximum vitality.
        3. age - the current age of the organism.
        4. lifespan - pre-defined maximum lifespan to be set based on scenario.
        5. fitness - how fit the organism is? Set to maximum fitness.
        6. death - reason of death (as death code).

    List of defined death codes
        1. death01 - zero vitality
        2. death02 - maximum age reached
```

```
    3. death03 - unknown death cause
"""
status = {'alive': True,                # is the organism alive?
          'vitality': 100.0,            # % of vitality
          'age': 0.0,                   # age of the organism
          'lifespan': 100.0,            # maximum lifespan
          'fitness': 100.0,             # % of fitness
          'death': None}

genome = []

def __init__(self, genome='default',
             mutation_type='point',
             additional_mutation_rate=0.01, gender=None):
    """
    Sets up a new organism with default status (age = 0, vitality = 100,
    lifespan = 100, fitness = 100, alive = True)

    @param genome: list of chromosomes to inherit. Default = 'default',
        which will set up one default chromosome. It also allows a
        'dummy' chromosome which is basically a one-base chromosome - this
        is for applications which does not utilize the chromosome.
    @param mutation_type: type of mutation. Accepts 'point' (point
        mutation), 'insert' (insert a base), 'delete' (delete a base),
        'invert' (invert a stretch of the chromosome), 'duplicate'
        (duplicate a stretch of the chromosome), 'translocate' (translocate
        a stretch of chromosome to another random position).
        Default = point.
    @param additional_mutation_rate: probability of mutation per base above
        background mutation rate. Default = 0.01 (1%). No mutation event
        will ever happen if (rate + background_mutation) is less than zero.
    @param gender: establishes the gender of the organism which may be used
        for mating routines.
    """
    if genome == 'default':
        self.genome = [Chromosome()]
    elif genome == 'dummy':
        self.genome = [Chromosome([0])]
    else:
        self.genome = genome
    self.mutation_type = mutation_type
    self.additional_mutation_rate = additional_mutation_rate
    self.gender = gender

def fitness(self):
    """
    Function to calculate the fitness of the current organism.
    B{This function MUST be over-ridden by the inherited class or
    substituted as fitness function is highly  dependent on utility.} The
    only requirement is that a fitness score must be returned.

    Here, the sample implementation calculates fitness as proportion of the
    genome with '1's.

    @return: fitness score or fitness list
    """
    one_count = sum([sum(chromosome.sequence)
                     for chromosome in self.genome])
    length_of_genome = sum([len(chromosome.sequence)
                            for chromosome in self.genome])
    return float(one_count) / float(length_of_genome)

def mutation_scheme(self, type=None, rate=None):
    """
```

```
    Function to trigger mutation events in each chromosome. B{This function
    may be over-ridden by the inherited class or substituted to cater for
    specific mutation schemes but not an absolute requirement to do so.}

    Both type and rate must be defined at the same time, otherwise the
    initiated mutation_type and additional_mutation_rate will be used.

    @param type: type of mutation. Accepts 'point' (point mutation),
        'insert' (insert a base), 'delete' (delete a base), 'invert'
        (invert a stretch of the chromosome), 'duplicate' (duplicate a
        stretch of the chromosome), 'translocate' (translocate a stretch of
        chromosome to another random position). Default = None.
    @param rate: probability of mutation per base above background mutation
        rate. Default = None. No mutation event will ever happen if
        (rate + background_mutation) is less than zero.
    """
    for chromosome in self.genome:
        if not type and not rate:
            chromosome.rmutate(self.mutation_type,
                               self.additional_mutation_rate)
        if type and rate:
            chromosome.rmutate(type, rate)

def setStatus(self, variable, value):
    """
    Sets new status or change status of the organism. However, the following
    status change will result in death of the organism
        1. 'alive' to False
        2. 'vitality' to or below zero
        3. 'age' to or above lifespan

    @param variable: name of status to change
    @param value: new value of the status
    """
    if variable == 'alive' and value == True:
        self.status['alive'] = value
    if variable == 'alive' and value == False:
        self.status['alive'] = value
        self.status['death'] = 'death03'
    if variable == 'vitality':
        if float(value) > 100.1:
            self.status['vitality'] = 100.0
        if float(value) > 0.0 and float(value) < 100.1:
            self.status['vitality'] = float(value)
        if float(value) == 0 or float(value) < 0:
            self.status['vitality'] = 0
            self.status['alive'] = False
            self.status['death'] = 'death01'
    elif variable == 'age':
        if float(value) < self.status['lifespan']:
            self.status['age'] = float(value)
        else:
            self.status['age'] = self.status['lifespan']
            self.status['alive'] = False
            self.status['death'] = 'death02'
    else:
        self.status[variable] = value

def getStatus(self, variable):
    """
    Returns a status variable of the current organism

    @param variable: name of the status variable
    @return: status or a KeyError if status is not found
```

```
        """
        try:
            return self.status[variable]
        except:
            raise KeyError('%s not found in organism status' % str(variable))

    def __str__(self):
        """Returns the genome of the organism"""
        return str([chromosome.sequence for chromosome in self.genome])

    def clone(self):
        """
        Cloness (deep copy) the organism.

        @return: a copy of current organism.
        """
        return deepcopy(self)


class Population(object):
    """
    Representation of a population as a list of organisms. The entire
    population is in memory.

    Methods to be over-ridden in the inherited class or substituted are
        - prepopulation_control
        - mating
        - postpopulation_control
        - generation_events
        - report
    """

    def __init__(self, goal, maxgenerations='infinite', agents=[]):
        """
        Establishes a population of organisms.

        @param goal: the goal to be reached by the population.
        @type goal: the return type of Organism.fitness()
        @param maxgenerations: maximum number of generations to evolve.
            Default = 'infinite'.
        @param agents: organisms making up the initial population.
        @type agents: list of Organism objects
        """
        self.agents = agents
        self.goal = goal
        self.maxgenerations = maxgenerations
        self.generation = 0

    def prepopulation_control(self):
        """
        Function to trigger population control events before mating event in
        each generation (For example, to simulate pre-puberty death). B{This
        function may be over-ridden by the inherited class or substituted to
        cater for specific events.} Although this is not an absolute
        requirement, it is extremely encouraged to prevent exhaustion of memory
        space. Without population control, it will seems like a reproducing
        immortal population.

        Here, the sample implementation eliminates the bottom half of the
        population based on fitness score unless the number of organisms is
        less than 20. This is to prevent extinction. However, if the number of
        organisms is more than 2000 (more than 100x initial population size,
        a random selection of 2000 will be used for the next generation.
        """
```

```
    size = len(self.agents)
    sfitness = [self.agents[x].fitness() for x in range(size)]
    threshold = sum(sfitness) / len(sfitness)
    temp = [self.agents[x]
            for x in range(size)
                if sfitness[x] > threshold]
    if len(temp) > 2001:
        size = len(temp)
        self.agents = [temp[random.randint(0, size - 1)]
                       for x in xrange(2000)]
    if len(temp) > 21:
        self.agents = temp

def mating(self):
    """
    Function to trigger mating events in each generation. B{This function
    may be over-ridden by the inherited class or substituted to cater for
    specific mating schemes but not an absolute requirement to do so.}
    Mating schemes should include
        - selection of mating partners using Organism.fitness() function
            and/or other status
        - processes and actions of mating

    Here, the sample implementation randomly selects any 2 organisms from
    the culled list and generate a new genome for the progeny organism by
    single crossover.
    """
    size = len(self.agents)
    temp = []
    for x in xrange(size):
        organism1 = self.agents[random.randint(0, size - 1)]
        organism2 = self.agents[random.randint(0, size - 1)]
        crossover_pt = random.randint(0, len(organism1.genome[0].sequence))
        (g1, g2) = crossover(organism1.genome[0], organism2.genome[0],
                             crossover_pt)
        temp = temp + [Organism([g1])]
    self.add_organism(temp)

def postpopulation_control(self):
    """
    Function to trigger population control events after mating event in
    each generation(For example, to simulate old-age death). B{This
    function may be over-ridden by the inherited class or substituted to
    cater for specific events.} Although this is not an absolute
    requirement, it is extremely encouraged to prevent exhaustion of memory
    space. Without population control, it will seems like a reproducing
    immortal population.
    """
    pass

def generation_events(self):
    """
    Function to trigger other defined events in each generation. B{This
    function may be over-ridden by the inherited class or substituted to
    cater for specific events but not an absolute requirement to do so.}
    Events and controls may include
        - processes simulating disaster or other catastrophic events
        - changes in mutations
    """
    pass

def report(self):
    """
    Function to report the status of each generation. B{This function may
```

```
    be over-ridden by the inherited class or substituted to cater for
    specific reporting schemes but not an absolute requirement to do so.}
    At the very least, this function should report whether the goal is
    reached.

    @return: dictionary of status describing the current generation
    """
    sfitness = [self.agents[x].fitness() for x in xrange(len(self.agents))]
    afitness = sum(sfitness) / float(len(self.agents))
    return {'generation': self.generation,
            'average fitness': afitness,
            '% to goal': float(afitness - self.goal) / self.goal * 100}

def generation_step(self):
    """
    Function to simulate events for one generation. These includes
        - mating (according to the mating scheme or function)
        - mutating each organism in the population
        - population size control
        - other events defined under generation_events function
        - increment of generation count
        - reporting the population status

    @return: information returned from report function.
    """
    if self.generation > 0:
        self.prepopulation_control()
    self.mating()
    self.postpopulation_control()
    for organism in self.agents:
        organism.mutation_scheme()
    self.generation_events()
    self.generation = self.generation + 1
    return self.report()

def add_organism(self, organism):
    """Add a new organism(s) to the population.

    @param organism: list of new Organism object(s)"""
    self.agents = self.agents + organism

def freeze(self, prefix='pop', proportion=0.01):
    """
    Preserves part or the entire population. If the population size or the
    preserved proportion is below 100, the entire population will be
    preserved. The preserved sample will be written into a file with name in
    the following format - <prefix><generation count>_<sample size>.gap

    @param prefix: prefix of file name. Default = 'pop'.
    @param proportion: proportion of population to be preserved.
        Default = 0.01, preserves 1% of the population.
    """
    import cPickle
    if proportion > 1.0: proportion = 1.0
    if len(self.agents) < 101 or len(self.agents) * proportion < 101:
        sample = self.agents
    else:
        size = len(self.agents)
        sample = [self.agents[random.randint(0, size - 1)]
                  for x in xrange(int(len(self.agents) * proportion))]
    name = ''.join([prefix, str(self.generation), '_',
                    str(len(sample)), '.gap'])
    f = open(name, 'w')
    cPickle.dump(sample, f)
```

```
        f.close()

    def revive(self, filename, type='replace'):
        """
        Revives a frozen population.

        @param filename: file name of frozen population (generated by
            Population.freeze() function)
        @param type: type of revival. Allows 'replace' (replace the current
            population with the revived population) or 'add' (add the revived
            population to the current population). Default = 'replace'.
        """
        import cPickle
        if type == 'replace':
            self.agents = cPickle.load(open(filename, 'r'))
        if type == 'add':
            self.agents = self.agents + cPickle.load(open(filename, 'r'))

def crossover(chromosome1, chromosome2, position):
    """
    Cross-over operator - swaps the data on the 2 given chromosomes after
    a given position.

    @param chromosome1: Chromosome object
    @param chromosome2: Chromosome object
    @param position: base position of the swap over
    @type position: integer
    @return: (resulting chromosome1, resulting chromosome2)
    """
    seq1 = chromosome1.sequence
    seq2 = chromosome2.sequence
    position = int(position)
    if len(seq1) > position and len(seq2) > position:
        new1 = Chromosome(seq1[:position] + seq2[position:],
                          chromosome1.base)
        new2 = Chromosome(seq2[:position] + seq1[position:],
                          chromosome2.base)
        return (new1, new2)
    elif len(seq1) > position:
        new1 = Chromosome(seq1[:position], chromosome1.base)
        new2 = Chromosome(chromosome2.sequence + seq1[position:],
                          chromosome2.base)
        return (new1, new2)
    elif len(seq2) > position:
        new1 = Chromosome(chromosome1.sequence + seq2[position:],
                          chromosome1.base)
        new2= Chromosome(seq2[:position], chromosome2.base)
        return (new1, new2)
    else:
        return (chromosome1, chromosome2)

population_data = \
{
    'nucleotide_list' : [1, 2, 3, 4],
    'chromosome_length' : 200,
    'chromosome_type' : 'defined',
    'chromosome' : [1] * 200,
    'background_mutation' : 0.0001,
    'genome_size' : 1,
    'population_size' : 200,
    'fitness_function' : 'default',
    'mutation_scheme' : 'default',
    'additional_mutation_rate' : 0.01,
    'mutation_type' : 'point',
```

```
    'goal' : 4,
    'maximum_generation' : 'infinite',
    'prepopulation_control' : 'default',
    'mating' : 'default',
    'postpopulation_control' : 'default',
    'generation_events' : 'default',
    'report' : 'default'
}


def population_constructor(data=population_data):
    """
    Function to construct a population based on a dictionary of population data.

    Population data contains the following keys:
        - 'nucleotide_list' = List of allowable nucleotides (bases).
          Default = [1, 2, 3, 4].
        - 'chromosome_length' = Length of a chromosome. Default = 200.
        - 'chromosome_type' = Type of chromosome. Default = 'defined'.
        - 'chromosome' = Initial chromosome. Default = [1] * 200.
        - 'background_mutation' = Background mutation rate.
          Default = 0.0001 (0.01%).
        - 'genome_size' = Number of chromosomes per organism. Default = 1.
        - 'population_size' = Size of initial population (number of organisms).
          Default = 200.
        - 'fitness_function' = Fitness evaluation function. Accepts 'default'
          or a function. Please refer to Organism. Default = 'default'.
        - 'mutation_scheme' = Function simulating the mutation scheme of an
          organism. Accepts 'default' or a function. Please refer to Organism.
          Default = 'default'.
        - 'additional_mutation_rate' = Mutation rate on top of background
          mutation rate. Default = 0.01 (1%).
        - 'mutation_type' = Type of default mutation. Default = 'point'.
        - 'goal' = Goal of the population, as evaluated by fitness function.
          Default = 4.
        - 'maximum_generation' = Number of generations to simulate. Accepts an
          integer or 'infinite'. Default = 'infinite'.
        - 'prepopulation_control' = Function simulating pre-mating population
          control. Please refer to Population. Default = 'default'.
        - 'mating' = Function simulating mating procedure (mate selection and
          act of mating control. Please refer to Population.
          Default = 'default'.
        - 'postpopulation_control' = Function simulating post-mating population
          control. Please refer to Population. Default = 'default'.
        - 'generation_events' = Function simulating other possible (usually
          rare or random) events in the generation. Please refer to
          Population. Default = 'default'.
        - 'report' = Function to generate the status report of the generation.
          Please refer to Population. Default = 'default'.

    @param data: population data
    @type data: dictionary

    @return: Population object
    """
    chr = Chromosome(data['chromosome'],
                     data['nucleotide_list'],
                     data['background_mutation'])
    org = Organism([chr]*data['genome_size'],
                   data['mutation_type'],
                   data['additional_mutation_rate'])
    if data['fitness_function'] != 'default':
        Organism.fitness = data['fitness_function']
    if data['mutation_scheme'] != 'default':
```

```
        Organism.mutation_scheme = data['mutation_scheme']
    org_set = [org.clone() for x in range(data['population_size'])]
    pop = Population(data['goal'],
                     int(data['maximum_generation']),
                     org_set)
    if data['prepopulation_control'] != 'default':
        Population.prepopulation_control = data['prepopulation_control']
    if data['mating'] != 'default':
        Population.mating = data['mating']
    if data['postpopulation_control'] != 'default':
        Population.postpopulation_control = data['postpopulation_control']
    if data['generation_events'] != 'default':
        Population.generation_events = data['generation_events']
    if data['report'] != 'default':
        Population.report = data['report']
    return pop

def population_simulate(population,
                        printfreq=100,
                        freezefreq='never',
                        freezefile='pop',
                        freezeproportion=0.01,
                        resultfile='result.txt'):
    """
    Function to simulate the population - start the GA.

    @param population: Population to run.
    @type population: Population object
    @param printfreq: Reporting intervals on screen. Default = 100.
    @param freezefreq: Generation intervals to freeze population into a file.
        See Population.freeze method. Accepts an integer or 'never'.
        Default = 'never'.
    @param freezefile: Prefix of file name of frozen population.
        See Population.freeze method. Default = 'pop'.
    @param freezeproportion: Proportion of population to be preserved.
        See Population.freeze method. Default = 0.01, preserves 1% of the
        population.
    @param resultfile: Name of file to print out results of each generation.
        Format of output is dependent on reporting method of the population
        (Population.report). Default = 'result.txt'.
    """
    if freezefreq == 'never': freezefreq = int(12e14)
    result = open(resultfile, 'w')
    report = population.generation_step()
    reportitems = ['|'.join([str(key), str(report[key])])
                   for key in report.keys()]
    result.writelines('|'.join(reportitems))
    result.writelines(os.linesep)
    while population.generation < population.maxgenerations:
        report = population.generation_step()
        reportitems = ['|'.join([str(key), str(report[key])])
                       for key in report.keys()]
        result.writelines('|'.join(reportitems))
        if population.generation % int(printfreq) == 0:
            print '|'.join(reportitems)
        result.writelines(os.linesep)
        if population.generation % int(freezefreq) == 0:
            population.freeze(freezefile, freezeproportion)
    population.freeze(freezefile, 1.0)
    result.close()
```

3. Example

In this example, we demonstrate the use of this module. We define a chromosome bearing the integer value of 1 to 4 at each base and the fitness of an individual as the average value of the chromosome where the maximum average value of 4 signifying the highest possible fitness, we examine the effects of the length of chromosome on the fitness level under identical mutation rates and scheme. The initial chromosome contains only '1' for the base. The background mutation rate is set at 0.01% with an additional mutation rate of 1%. Only point mutation and 200 individuals are used in this example. We examined chromosome length between 200 to 1400 bases for 5000 generations using the following code:

```
import genetic as g

pdata = \
{
    'nucleotide_list' : [1, 2, 3, 4],
    'chromosome_length' : 200,
    'chromosome_type' : 'defined',
    'chromosome' : [1] * 200,
    'background_mutation' : 0.0001,
    'genome_size' : 1,
    'population_size' : 200,
    'fitness_function' : 'default',
    'mutation_scheme' : 'default',
    'additional_mutation_rate' : 0.01,
    'mutation_type' : 'point',
    'goal' : 4,
    'maximum_generation' : 5000,
    'prepopulation_control' : 'default',
    'mating' : 'default',
    'postpopulation_control' : 'default',
    'generation_events' : 'default',
    'report' : 'default'
}

for chromosome_length in range(200, 1600, 100):
    pdata['chromosome'] = [0] * chromosome_length
    pop = g.population_constructor(pdata)
    g.population_simulate(pop, 100, 'never', 'pop', 0.1,
                          str(chromosome_length)+'result.txt')
```

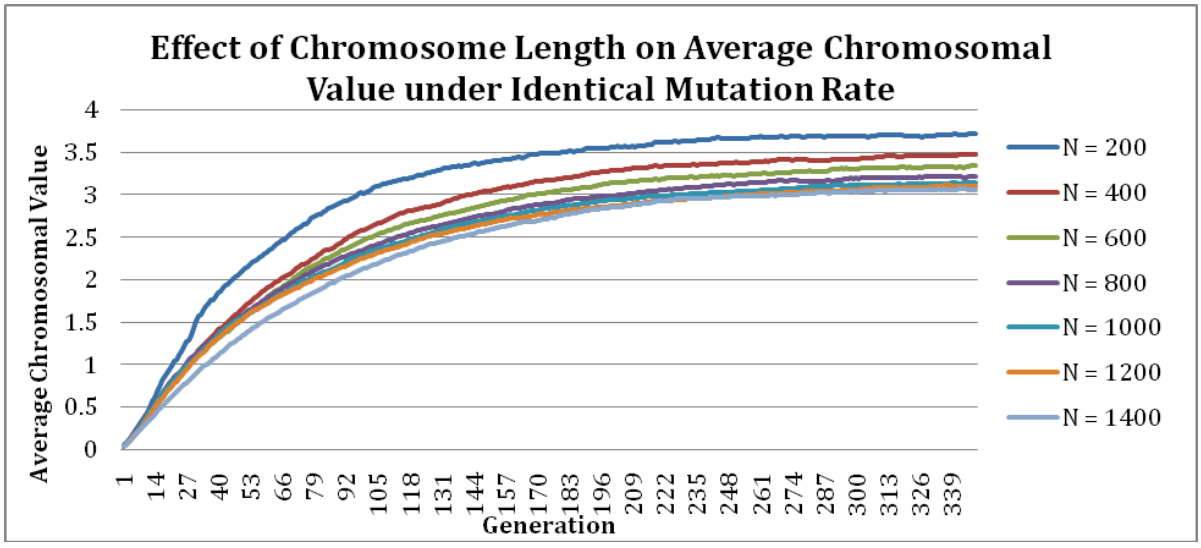

Figure 1: Effect of chromosome length from 200 bases to 1400 bases on average chromosomal value within 350 generations.

Our results (Figure 1) suggested that the fitness of an individual, calculated at the average value of its chromosome, trends towards the average value of 2.5.

4. References

ElNainay, MY, Ge, F, Wang, Y, Hilal, AE, Shi, Y, MacKenzie, AB and Bostian, CW. 2009. Channel allocation for dynamic spectrum access cognitive networks using localized island genetic algorithm. 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, pp. 1-3.

Engelbrecht, AP. 2007. Computational intelligence: an introduction. John Wiley & Sons, Ltd.

Fraser, AS. 1957a. Simulation of genetic systems by automatic digital computers I: Introduction. Australian Journal of Biological Sciences 10:484-491.

Fraser, AS. 1957b. Simulation of genetic systems by automatic digital computers II: Effects of linkage on rates of advance under selection. Australian Journal of Biological Sciences 10:492-499.

Holland, JH. 1975. Adaptation in natural and artificial systems. University of Michigan Press.

King, WT, Jr. 2004. Using smaller populations to increase diversity in an island genetic algorithm. Master's Thesis. Graduate Faculty, University of South Alabama.

Melanie, M. 1999. An introduction to genetic algorithms. MIT Press.

Sheikh, RH, Raghuwanshi, MM and Jaiswal, AN. 2008. Genetic algorithm based clustering: a survey. First International Conference on Emerging Trends in Engineering and Technology, pp.314-319.

Skolicki, Z. 2005. An analysis of island models in evolutionary computation. Proceedings of the 2005 workshops on Genetic and evolutionary computation, pp. 386 - 389.

Srinivas, M and Patnaik, LM. 1994. Genetic algorithms: a survey. Computer 27(6): 17 – 26.

Whitley, D. 1994. A genetic algorithm tutorial. Statistics and Computing 4(2): 65-85.